Attorney's Docket No.: 10559-268001

AF

JFW

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant	: Debra Bernstein et al.	Art Unit	: 2124
Serial No.	: 09/747,019	Examiner	: Satish Rampuria
Filed	: December 21, 2000		
Title	: BREAKPOINT METHOD FOR PARALLEL HARDWARE THREADS IN MULTITHREADED PROCESSOR		

MAIL STOP APPEAL BRIEF – PATENTS
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF ON BEHALF OF
DEBRA BERNSTEIN, SERGE KOMFELD, DESMOND R. JOHNSON,
DONALD HOOPER, JAMES D. GUILFORD, AND RICHARD D. MURATORI

The Appeal Brief fee of **$500** is enclosed. Please apply any other charges or credits to Deposit Account No. 06-1050, referencing attorney docket No. 10559-268001.

09/27/2005 ZJUHAR1 00000017 09747019

01 FC:1402 500.00 OP

CERTIFICATE OF MAILING BY FIRST CLASS MAIL

I hereby certify under 37 CFR §1.8(a) that this correspondence is being deposited with the United States Postal Service as first class mail with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Sept. 23, 2005
Date of Deposit

Deborah R. Nast
Signature

Deborah R. Nast
Typed or Printed Name of Person Signing Certificate

(i.) Real Party In Interest

The real party in interest in the above application is Intel Corporation.

(ii.) Related Appeals and Interferences

The appellant is not aware of any appeals or interferences related to the above-identified patent application.

(iii.) Status of Claims

This is an appeal from the decision of the Primary Examiner in an Office Action dated May 6, 2005, rejecting claims 22-33, all of the claims of the above application. Claims 1-21 were canceled. The claims have been twice rejected. Claims 22-33 are the subject of this appeal.

(iv.) Status of Amendments

All amendments have been entered. Appellant has filed herewith a Notice of Appeal on **August 8, 2005**.

(v.) Summary of Claimed Subject Matter

Background

The claimed invention relates to techniques for facilitating debugging thread execution in a multiprocessor machine using breakpoints. [Specification page 1, lines 2-3]

In a parallel processor, many threads can execute simultaneously. [Specification page 1, lines 15-16] As a result, it can be difficult to isolate a cause (or causes) of error when several threads are executing simultaneously. The debugging problem is compounded in a processor architecture in which a parallel processor includes several microengines, each of which is capable of executing several threads simultaneously.

Appellant's Invention

Claim 22

One aspect of Appellant's invention is set out in claim 22, as a computer-implemented method. "Referring to FIG. 5, a breakpoint process 600 for selectively stopping parallel hardware threads from a remote user interface..." [Specification page 15, lines 4-6]

The inventive features of Appellant's claim 22 include, in parallel hardware threads executing in a processor comprising a plurality of microengines, receiving a source code line to be break pointed in a selected microengine. Appellant's FIG. 4 shows a multiprocessor 500 that includes a plurality of microengines 22a-f. [FIG. 4].

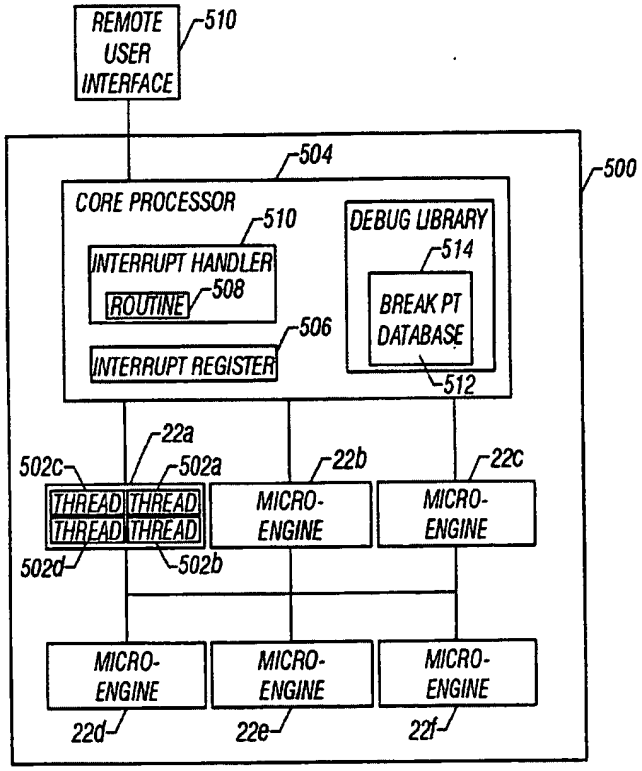

FIG. 4

Each of the microengines 22a-f is capable of processing in parallel multiple hardware-supported threads. [Specification page 4, lines 7-9, and page 10, lines 16-18]. As can be seen in

FIG. 4, microengine 22a has four threads, 502a-d, executing on that microengine. Also, "The core processor 504 is linked to a remote user interface 511. A user (not shown) inputs a source code line to be breakpointed in one of the microengines using the remote user interface 511." [Specification page 12, lines 5-9]

Another inventive feature of Appellant's claim 22 includes determining whether the source code line can be break pointed. FIG. 6, shown below, is a flowchart of the breakpoint process. "The central processor 504 searches a breakpoint database 512 in a debug library 514 to determine whether the instruction corresponding to the source code line can be breakpointed. Since there are certain cases where breakpointing is not allowed, i.e., a trap in the code cannot be inserted at this position in the code to signal a breakpoint.. For example, if two instructions must be executed in succession, i.e., a register of one is used in the very next cycle in the next, the first instruction cannot be breakpointed. This is because the software breakpoint inserts a branch to displace the breakpointed instruction, thus separating the two instructions." [Specification page 12, lines 5-20]

An inventive feature of Appellant's claim 22 also includes identifying, if the source line code can be break pointed, the selected microengine to insert a break point into, which microengine threads to enable breakpoints for, and which microengines to stop if a breakpoint occurs. "After... determining that an instruction may be breakpointed, the core processor 504 invokes a remote procedure call (RPC) referred to as SetBreakPoint to the Debug library 514." [Specification page 12, lines 21-24, and see also 608 in FIG. 5 and page 15, lines 10-12] "The SetBreakpoint RPC identifies which microengine to insert the breakpoint into, which program counter (PC) to breakpoint at, which microengine threads (referred to as contexts) to enable breakpoint for, and which microengines to stop if the breakpoint occurs." [Specification page 12, line 24 to page 13, line 5]

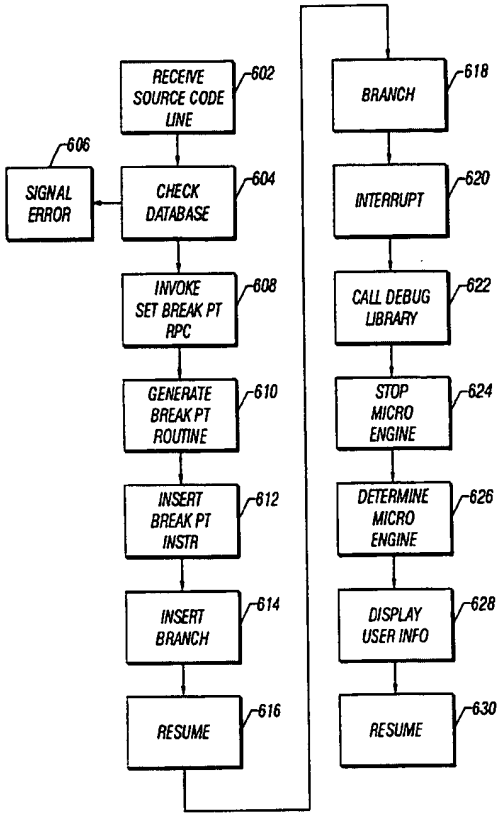

FIG. 5

Another inventive feature of Appellant's claim 22 includes signaling an error if the source code line cannot be breakpointed. "The process 600 [shown in FIG. 5] fetches data 604 within its database to determine whether the source code line in the microengine may be breakpointed. If not, the process 600 signals 606 an error to the user." [Specification page 15, lines 7-10].

Claim 28

Claim 28 recites another aspect of the invention. Claim 28 is a processor that can execute multiple parallel threads in multiple microengines [FIG. 4, and Specification page 4, lines 7-9, and page 10, lines 16-18].

Inventive features of Appellant's claim 28 include a register stack, and a program counter for each executing context. "Referring to FIG. 2, an exemplary one of the microengines, microengine 22f, is shown." [Specification page 7, lines 14-15].

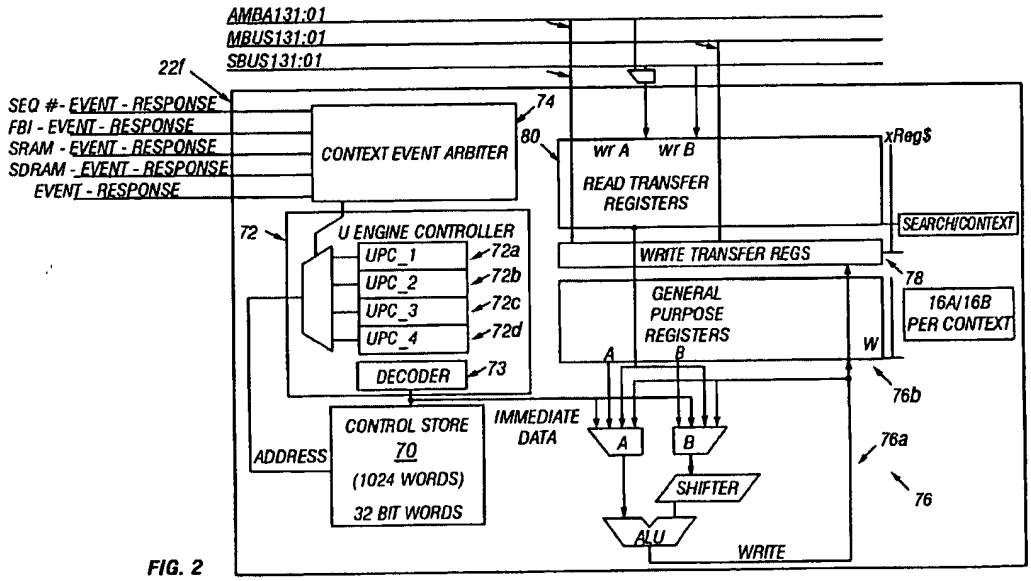

FIG. 2

"The microengine 22f also includes a write transfer register stack 78 and a read transfer stack 80. These registers 78 and 80 are also windowed so they are relatively and absolutely addressable. The write transfer register stack 78 is where write data to a resource is located. Similarly, the read register stack 80 is for returned data from a shared resource." [Specification page 9, lines 7-13]. "The microengine 22f also includes controller logic 72. The controller logic 72 includes in instruction decoder 73 and program counter units 72 a-d." [Specification page 7, lines 19-21].

Another inventive feature of Appellant's claim 28 includes an arithmetic logic unit coupled to the register stack. As shown above in FIG. 2, the ALU is coupled to the write transfer register stack 78, the read transfer register stack 80, and the general purpose register stack 76b. [FIG. 2].

Another inventive feature of Appellant's claim 28 includes a program control store that stores a breakpoint routine. "The microengine 22f includes a control store 70, which, in an implementation includes a RAM of here 1,024 words of 32-bits each." [Specification page 7, lines 15-17] Also, "[a] processor comprises an arithmetic logic unit coupled to the register stack and a program control store that stores a breakpoint command." [FIG. 4 shows a routine 508 inside interrupt handler 510, and original claim 11 and original claim 12 support the above feature]

Claim 28 also includes the breakpoint routine causes the processor to receive a source code line to be break pointed in a selected microengine. [Specification page 12, lines 5-9, and page 15, lines 4-7] The breakpoint routine also causes the processor to determine whether the source code line can be break pointed. [Specification page 12, lines 5-20, and page 15, lines 7-9]. The breakpoint routine also causes the processor to identify, if the source line code can be break pointed, the selected microengines to insert a break point into, which microengine threads to enable breakpoints for, and which microengines to stop if a breakpoint occurs. [Specification page 12, line 21 to page 13, line 5, and page 15, lines 10-16]. And, the breakpoint routine also causes the processor to signal an error if the source code line cannot be break pointed. [Specification page 15, lines 7-10].

(vi.) Ground of Rejection to be Reviewed on Appeal

Claims 22-33 stand rejected under 35 U.S.C. 103(a) as being unpatentable over U.S. Patent 6,378,125 to Bates in view of U.S. Patent 5,815,714 to Shridhar et al.

(vii.) Argument

Obviousness

"It is well established that the burden is on the PTO to establish a prima facie showing of obviousness, *In re Fritsch*, 972 F.2d. 1260, 23 U.S.P.Q.2d 1780 (C.C.P.A., 1972)."

"It is well established that there must be some logical reason apparent from the evidence or record to justify combination or modification of references. *In re Regal*, 526 F.2d 1399 188, U.S.P.Q.2d 136 (C.C.P.A. 1975). In addition, even if all of the elements of claims are disclosed

in various prior art references, the claimed invention taken as a whole cannot be said to be obvious without some reason given in the prior art why one of ordinary skill in the art would have been prompted to combine the teachings of the references to arrive at the claimed invention. Id. Even if the cited references show the various elements suggested by the Examiner in order to support a conclusion that it would have been obvious to combine the cited references, the references must either expressly or impliedly suggest the claimed combination or the Examiner must present a convincing line of reasoning as to why one skilled in the art would have found the claimed invention obvious in light of the teachings of the references. *Ex Parte Clapp*, 227 U.S.P.Q.2d 972, 973 (Board. Pat. App. & Inf. 985)."

"The mere fact that the prior art could be so modified would not have made the modification obvious unless the prior art suggested the desirability of the modification." *In re Gordon*, 221 U.S.P.Q. 1125, 1127 (Fed. Cir. 1984).

> Although the Commissioner suggests that [the structure in the primary prior art reference] could readily be modified to form the [claimed] structure, "[t]he mere fact that the prior art could be so modified would not have made the modification obvious unless the prior art suggested the desirability of the modification." *In re Laskowski*, 10 U.S.P.Q. 2d 1397, 1398 (Fed. Cir. 1989).

"The claimed invention must be considered as a whole, and the question is whether there is something in the prior art as a whole to suggest the desirability, and thus the obviousness, of making the combination." *Lindemann Maschinenfabrik GMBH v. American Hoist & Derrick*, 221 U.S.P.Q. 481, 488 (Fed. Cir. 1984).

> Obviousness cannot be established by combining the teachings of the prior art to produce the claimed invention, absent some teaching or suggestion supporting the combination. Under Section 103, teachings of references can be combined only if there is some suggestion or incentive to do so. *ACS Hospital Systems, Inc. v. Montefiore Hospital*, 221 U.S.P.Q. 929, 933 (Fed. Cir. 1984) (emphasis in original, footnotes omitted).

"The critical inquiry is whether 'there is something in the prior art as a whole to suggest the desirability, and thus the obviousness, of making the combination.'" *Fromson v. Advance Offset Plate, Inc.*, 225 U.S.P.Q. 26, 31 (Fed. Cir. 1985).

The examiner has mischaracterized the teachings of Bates et al., U.S. Patent 6,378,125 and Shridhar et al., U.S. Patent 5,815,714 in rejecting Claims 22-33.

Claims 22, 24, 26, 27, 28, 30, 32, 33

For the purposes of this appeal only, claims 22, 24, 26, 27, 28, 30, 32, and 33 may be treated as standing or falling together. Claim 22 is representative of this group.

Claim 22 is directed to a computer-implemented method that requires "determining whether the source code line can be break pointed. The examiner admits that "Bates does not explicitly disclose determining whether the source code line can be break pointed" (Office Action page 3). The examiner, however, contends that Shridhar (Col. 6, lines 19-21) discloses this feature by "decoder... determines... if there is an embedded debug command present in the source code line." (Office Action page 3).

The Examiner's characterization of what Shridhar discloses is not correct. Shridhar does not disclose determining whether the source code line can be break pointed. Rather, Shridhar discloses a method and apparatus for re-generating debug commands from a source program having embedded debug commands in a comment field. The debug commands taught by Shridhar are already in the code, and it is therefore unnecessary to make any determination of whether source code line can be breakpointed.

As shown in Shridhar's FIG. 2, a debug command 208 is embedded into the comments field 204 (Col. 4, lines 31-37). Shridhar describes an assembler 400 that is used to process a user-developed source code that includes embedded debug commands. The assembler 400 comprises a decoder 402, a command processor 404, a memory module 406, a command file generator 408, and object code generator 410 (Col. 5, lines 55-59). During the debugging of the user-developed source code, the source code is loaded into the assembler 400, and the decoder 402 starts reading in one line of code at a time from the user-developed source code (Col. 6, lines 16-19). As Shridhar describes (column 6, lines 16-26):

> **The decoder 402 initiates the extraction processes checking (503) to see if the source file end has been reached, and if not reads (504) in a first (or next) line of code for processing. The decoder 402 determines (505) if there is an embedded debug command present in the source code line, and if not assembles (506) the line of code and returns to read in the next line of code at step 504. If an embedded debug command is present the decoder 402 extracts the debug commands from the source code for processing by the command processor 404 a single command at a time.**

Thus, contrary to the examiner's assertion, Shridhar neither describes nor suggests to determine whether the source code line can be break pointed. Rather, Shridhar merely checks to see if a debug command has been embedded in the source code line being decoded. In other words, Shridhar's assembler 400 does not itself determine if it is permissible to insert a break point at a particular source code line because in the system taught by Shridhar the debug commands are already embedded in the source code prior to the debugging process. Accordingly, Shridhar neither describes nor suggests "determining whether the source code line can be break pointed," as recited in claim 22.

Claim 22 recites the additional distinct feature that "if the source line can be break pointed, identifying the selected microengine to insert a break point into." Here also, the examiner admits that "Bates does not explicitly disclose"...if the source line can be break pointed, identifying the selected microengine to insert a break point into" (Office Action page 3). The examiner, however, relies upon Shridhar to teach this feature by "[i]f there is a "HALT" at the end of debug command," and "determine the type of debug command... generates... break points" (Office Action page 3).

Here again, the examiner's characterization of Shridhar is incorrect. Shridhar's assembler 400 facilitates the debugging of a single user-developed source program. The debugging of the user developed source program is performed in a single processor environment. As described by Shridhar (col. 5, lines 57-66):

> **The assembler 400 comprises a decoder 402, a command processor 404, a memory module 406, a command file generator 408 and object code generator 410. The decoder 402 extracts the embedded debug commands from the source program and passes the embedded commands to the command processor 404. The command processor 404 determines the type of debug command and either stores the command temporarily in the memory module 406 or generates an appropriate break point command for inclusion in the command file.**

Thus, Shridhar always stores the debug command in the same memory module 406 (shown in FIG. 4). Shridhar does not contemplate a multiprocessor environment, in which the user-developed source program would execute, and therefore Shridhar is not concerned with identifying the selected microengine to insert a breakpoint into. Rather, in Shridhar there are no microengines from which to select. Accordingly, Shridhar neither describes nor suggests, nor is it even relevant, to the feature that "if the source code line can be break pointed, identifying the selected microengine to insert a break point into," as recited in claim 22.

Claim 22 adds the further distinct feature of "if the source line can be break pointed, identifying ... which microengine threads to enable breakpoints for, and which microengines to stop if a break point occurs." The examiner admits that "Bates does not explicitly disclose ...if the source line can be break pointed, identifying ... which microengine threads to enable break points for, and which microengines to stop if a breakpoint occurs" (Office Action page 3). The examiner, however, at page 3 of the Office Action, relies, upon Shridhar for teaching this feature by "[i]f there is a "HALT" at the end of debug command", and by "Break point commands... "HALT" commands, which results in the termination of the simulation... and "CONT" commands which perform a debug function as directed..." (col. 5, lines 17-26).

Again, the examiner's characterization of Shridhar is incorrect. Shridhar explains (Col. 5, lines 15-26):

> **Break-point commands can be categorized as being of two major types: 1) "HALT" commands, which result in the termination of the simulation (and necessarily the execution of the object code) due to overflow conditions or other serious errors; and finally 2) "CONT" commands, which perform a debug function as directed by the individual command, then continue execution of the object code by the simulator where the break occurred.**

Shridhar further explains (Col. 6, lines 43-63):

> **The command processor 404 checks (509) to see if the extracted debug command ends in a "HALT." If there is a "HALT" at the end of the debug command, the command processor 404 will generate (510) a break-point command with a "halt" at the end of it. This command is written (513) to the command file and includes the relative address in memory associated with the position of the debug command in the source program. Those**

> **ordinarily skilled in the art will recognize that until the object code has been loaded into a designated address space in memory by the simulator loader, no actual address locations exist, and all addressing is only relative to a theoretical base address at which the source code originates. The "HALT" command acts to stop the execution of the object program and often is used to indicate overflow or other serious error conditions.**
>
> **Finally, if the debug command being processed does not contain a "HALT" at the end of the command, then it is of the "CONT" (continue) type and the command processor 404 will generate (512) a break-point command and automatically append a "CONT" command at the end of the debug command.**

While it is true that Shridhar's assembler 400 is able to identify the type of break-point command embedded in the user-developed source program (i.e., either a "HALT" command or a "CONT" command), the examiner is confusing that functionality as being the same as identifying which microengine threads to enable breakpoint for. The claim 22 feature is not directed to identifying types of break points, but to identifying which microengine and which thread to insert a break point into.

As noted above, the debugging of the user-developed source program is performed by Shridhar's assembler 400 in a single processor environment. At no point does Shridhar disclose or suggest a multi-processor environment, and certainly not a multi-microengine environment, in which the debugging process is performed. Nor does Shridhar disclose or suggest a multi-tasking processing environment in which multiple threads execute on a processor. Accordingly, Shridhar neither describes nor suggests, nor is Shridhar even relevant to the feature of: "if the source code line can be break pointed, identifying ... which microengine threads to enable breakpoint for, and which microengines to stop if a break point occurs," as recited in claim 22.

Claim 22 also includes the distinct feature of "if the source code line cannot be break pointed, signaling an error." The examiner admits that "Bates does not explicitly disclose ...if the source line cannot be break pointed, signaling an error" (Office Action page 3). The examiner, however, contends, at page 3 of the Office Action, that Shridhar discloses this feature by "HALT commands...termination of simulation...due to... serious error" (col. 5, lines 20-22).

The examiner's characterization of Shridhar with respect to this feature is also incorrect. Shridhar describes (Col. 5. lines 18-22):

> **Break-point commands can be categorized as being of two major types: 1) "HALT" commands, which result in the termination of the simulation (and necessarily the execution of the object code) due to overflow conditions or other serious errors;**

Appellant contends that Shridhar describes that a HALT break-point command causes the termination of the simulation when overflow conditions, or some other serious errors occur. However, for the HALT break-point command to terminate the simulation, the HALT command would first have to be placed in the source code. Shridhar describes that the HALT break-point command is inserted into the source code line (indeed, these commands have been inserted by the user at the time he wrote the source code). This is the opposite to what is contemplated by the feature of "if the source code line cannot be break pointed, signaling an error", in which the signaling of an error is the result of not being able to insert a breakpoint command into the source code line. Accordingly, Shridhar neither discloses nor suggests "if the source code line cannot be break pointed, signaling an error", recited by claim 22.

Claim 22 also includes the distinct feature of, in parallel hardware threads executing in a processor comprising a plurality of microengines, receiving a source code line to be break pointed in a selected microengine. The examiner contents that Bates describes that feature. Bates describes at col. 4, lines 9-60:

> **Referring to FIG. 2, an exemplary software environment is shown for the computer system 10 of FIG. 1. Specifically, thread identification capability is illustrated in block diagram form, with the elements shown that contribute to maintaining (e.g., creating and deleting) control points and to responding to a system exception. The debug user interface 24, which may be a third-party debugging program, is shown initiating the process, providing at Phase 1 any control points to be established. For example, a debugger command is made setting a thread identification control point or a break point at a given statement number or a variable.**

While Bates enables a user to set a control point at a particular thread, statement number or variable, Bates does not suggest that the user can specify a microengine. Accordingly, Bates does not disclose or suggest "in parallel hardware threads executing in a processor comprising a plurality of microengines, receiving a source code line to be break pointed in a selected microengine", recited by claim 22.

Claims 23 and 29

For the purpose of this appeal only, claims 23 and 29 may be treated as standing or falling together. Claim 23 is representative of this group

Claim 23 adds the distinct feature that identifying further comprises generating a break point routine by modifying a template of instructions stored in a debug library. The examiner contends Bates in Col. 7 lines 34-37 shows this feature. Bates discusses:

> **The break point manager routine 72 discussed above would be performing the corresponding actions with regard to the break point table 32, as discussed for FIG. 4. (Col. 7, lines 34-37)**

Bates discusses the aforementioned break point manager routine 72 at col. 5, lines 40-65, where Bates explains that:

> **Referring to FIG. 4, a break point manager routine 72 is illustrated wherein two types of break point operations are handled, based on whether the input is due to a break point (i.e., responding to a system exception) or a debugger command (block 74). If the input was not a hit break point in block 74, then a determination is made as to whether a control point is to be set (block 76). If a control point is be set, then the program element for which a control point is to be associated is mapped to a physical address (block 78). Then determination is made as to whether this address is already in the break point table (block 80). Referring to FIG. 3, this means searching each address field 48 to see if the address is included in an address field 48. Returning to FIG. 4, if the address is found in block 80, then routine 72 is done. If, however, the address is not currently in the break point table 32, then a record 47 is added to the break point table 32 (block 82). This record 47 will include data as to whether the control point is to be a thread identification control point, such as by setting high the thread ID control point flag 54 in FIG. 3. Next, the control point is set in the computer program 20. In the illustrative embodiment, setting the control point means inserting an invalid instruction at the address of the statement (block 84) while retaining the original op code in the op code field 50 of the break point table 32. The break point routine 72 is then done for this operation.**

This teaching in Bates does not suggest the feature of claim 23. It does not suggest generating a break point routine by modifying a template of instructions stored in a debug library.

The above paragraph in Bates describes a break point manager routine that performs break point operations. Bates discloses that the break point manager routine modifies the break

point table that maintains records holding information regarding the addresses of break points in threads. Nowhere does Bates disclose or suggest that the break point manager routine modifies a template of instructions. Rather, in Bates, the break point routine is an existing procedure that controls the handling of break points. Bates does not generate a break point routine by modifying a template of instructions. Accordingly, Bates does not disclose or suggest "generating a break point routine by modifying a template of instruction stored in a debug library," recited by claim 23.

Claim 25 and 31

For the purpose of this appeal only, claims 25 and 31 may be treated as standing or falling together. Claim 25 is representative of this group

Claim 25 adds the distinct feature of executing the break point routine, the break point routine stopping selected threads and determining which microengine sent an interrupt. The examiner contends that Bates at col. 6, line 44, col. 5, lines 6-7, and lines 15-16, shows this feature. Bates describes at col. 5, lines 6-17:.

> **After the control points are set, user provides an input that resumes execution of the program 20. As represented at Phase 6, execution of the program results in an encounter of a control point. In the illustrative embodiment, this is accomplished by an invalid statement in the decode program causing a system exception, similar to a break point. An interrupt handler, or similar means, passes information regarding the exception or interrupt to the break point manager 30. The break point manager 30 references and updates the break point table 32 at Phase 7 as required in order to determine what type of control point was encountered and the associated processing.**

The examiner's characterization of Bates is again incorrect. Bates determines the nature of the control point encountered (e.g., determine whether the control point was a user-inserted invalid instruction that triggered a hardware interrupt). Bates also determines at which of the executing threads the control point encountered is located. However, Bates does not describe or suggest determining which microengine sent the interrupt. Accordingly, Bates does not disclose or suggest, nor is relevant to "executing the break point routine, the break point routine stopping selected threads and determining which microengine sent an interrupt," recited by claim 25.

Conclusion

Appellant submits, therefore, that Claims 22-33 are allowable over the cited art. Therefore, the Examiner erred in rejecting Appellant's claims and should be reversed.

Respectfully submitted,

Date: Sept. 23, 2005

Ido Rabinovitch
Reg. No. L0080

PTO Customer No. 26161
Fish & Richardson P.C.
Telephone: (617) 542-5070
Facsimile: (617) 542-8906

21172843.doc

Appendix of Claims

1-21. (Cancelled)

22. A computer-implemented method comprising:

in parallel hardware threads executing in a processor comprising a plurality of microengines, receiving a source code line to be break pointed in a selected microengine;

determining whether the source code line can be break pointed;

if the source code line can be break pointed, identifying the selected microengine to insert a break point into, which microengine threads to enable breakpoints for, and which microengines to stop if a break point occurs; and

if the source code line cannot be break pointed, signaling an error.

23. The computer-implemented method of claim 22 wherein identifying further comprises generating a break point routine by modifying a template of instructions stored in a debug library.

24. The computer-implemented method of claim 23 wherein identifying further comprises inserting a break point at the source code line and a branch to the source code line.

25. The computer-implemented method of claim 24 further comprising:

executing the parallel hardware threads until the break point is encountered; and

executing the break point routine, the break point routine stopping selected threads and determining which microengine sent an interrupt.

26. The computer-implemented method of claim 25 further comprising displaying program information to a user.

27. The computer-implemented method of claim 26 further comprising resuming execution of the parallel threads in response to a user input.

28. A processor that can execute multiple parallel threads in multiple microengines and that comprises:

a register stack;

a program counter for each executing context;

an arithmetic logic unit coupled to the register stack and a program control store that stores a breakpoint routine that causes the processor to:

receive a source code line to be break pointed in a selected microengine;

determine whether the source code line can be break pointed;

if the source code line can be break pointed, identify the selected microengine to insert a break point into, which microengine threads to enable breakpoints for, and which microengines to stop if a break point occurs; and

if the source code line cannot be break pointed, signal an error.

29. The processor of claim 28 wherein identifying further comprises generating a break point routine by modifying a template of instructions stored in a debug library.

30. The processor of claim 29 wherein identifying further comprises inserting a break point at the source code line and a branch to the source code line.

31. The processor of claim 30 wherein the breakpoint routine further causes the processor to:

execute the parallel threads until the break point is encountered; and

stop selected threads and determine which microengine sent an interrupt.

32. The processor of claim 31 wherein the breakpoint routine further causes the processor to display program information to a user.

33. The processor of claim 32 the breakpoint routine that causes the processor to resume execution of the parallel threads in response to a user input.

EVIDENCE APPENDIX

None

RELATED PROCEEDINGS

APPENDIX

None